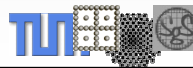


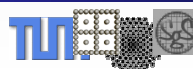
Implementation: Parallelization

- classical programming paradigms are, in principle, all well-suited for explicit or implicit parallelization:
 - **imperative**: FORTRAN, C (dominant male, recently with some OO-touch like in C++)
 - **logical/relational**: PROLOG
 - **object-oriented**: SMALLTALK
 - **functional/applicative**: LISP
- implicit parallelization typically via special compilers
- explicit parallelization typically via linked communication libraries
- traditional way ☹ in Scientific Computing: FORTRAN code, vectorizing compiler, CRAY, wait for results
- explicit parallelization often difficult (cf. Gauß-Seidel), this makes non-conventional approaches attractive



The Programming Model MPI 1

- How to write parallel programs?
 - UMA systems: simple answer – just as sequential ones
 - distributed memory systems: MPI model or standard
 - *Message Passing Interface*
 - originally for clusters (NOWs), today used even on massively parallel computers, too
 - MPI-1 developed 1992-1994
 - explicit exchange of messages: higher amount of programming work, but increasing possibilities of tuning and optimizing
 - MPI features:
 - parallel program: n processes, separate address spaces, no remote access
 - message exchange via system calls `send` and `receive`
 - MPI-kernel: library of communication routines, allowing to integrate MPI commands into standard languages



MPI 2

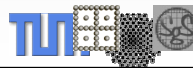
➤ MPI features (continued):

- messages consist of a *header* (recipient, buffer, type, context of communication) and of their *body* (contents)
- messages are buffered (send buffer, receive buffer)
- sending a message can be *blocking* (finished only after message has left node) or *non-blocking* (finished immediately, message may be sent later)
- the same holds for receiving a message (blocking: waiting; non-blocking: looking for it from time to time)

➤ cost of passing a message (length N, buffer cap. K):

$$t(N) = \alpha \cdot \frac{N}{K} + \beta \cdot N$$

initializing cost/time α , transportation cost β



Programming with MPI

➤ a simple example:

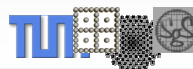
P1: compute something	P2: compute something
store result in SBUF	store result in SBUF
SendBlocking(P2,SBUF)	SendBlocking(P1,SBUF)
RecBlocking(P2,RBUF)	RecBlocking(P1,RBUF)
read data in RBUF	read data in RBUF
compute again	compute again

➤ without buffering: deadlocks possible

- nothing specified: buffering possible, but not imperative
- never: no buffering (efficient, but risky)
- always: secure, but sometimes costly

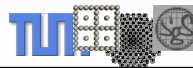
➤ collective communication features available:

- broadcast, gather, gather-to-all, scatter, all-to-all, ...



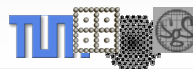
Load Distribution

- load: amount of work on processors
 - optimum: minimize idle times; needs estimates and monitoring
 - strategy: **load balancing** or **load distribution** or **scheduling**
 - important: avoid overhead
- one distinguishes
 - **global** (where do which processes run) and **local** (when does which processors which process) **scheduling**
 - **static** (a priori) and **dynamic** (during runtime) **load balancing**
- in Scientific Computing applications
 - load often not predictable (adaptive refinement of a finite element mesh, convergence behaviour of iterations may differ)
 - thus: static load balancing not sufficient



Designing Load Distribution

- Which are the primary objectives?
 - optimization of *system load* or *application runtime*?
 - *placement* of new processes or *migration* of running processes?
- Which is the level of integration?
 - Who initiates actions (measure load, chose strategy, take measures) – application program / runtime system / OS?
- Any special features of the application to be considered?
 - restrictions in allocation process-to-processor frequent in S.C.
- Which units shall be distributed or displaced?
 - whole processes (coarse grain)
 - threads (fine grain)
 - objects or data (typical for simulation applications)

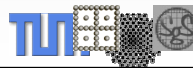


Classification of Strategies

- origin of the idea:
 - from physics (diffusion model), from combinatorics (graph theory), economics (bidding, brokerage)
 - for networks, for bus topologies
 - data represented as grids, trees, sets, or ...
- distribution mechanisms:
 - load handed over to neighbouring nodes only?
 - just distribution of new units or migration of running ones (how?)?
- flow of information:
 - to whom is load communicated, from where comes information?
- coordination:
 - who makes decisions? autonomous/cooperative/competitive?
- algorithms:
 - who initiates measures? adaptivity? costs relevant? evaluation?



Introduction to Scientific Computing
Lesson 11: Parallelization



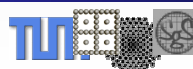
Slide 7

Examples of LD-Strategies

- diffusion model:
 - permanent balancing process between neighbours
- bidding model:
 - supply and demand, establishment of some market
- broker model:
 - esp. for heterogeneous hierarchical topologies, scalable
 - broker with partial knowledge, budget-based decision whether local processing or looking for better offers
 - prices for use of resources and brokerage
- matching model:
 - construct matching in topology graph, balance along edges
- balanced allocation, space-filling curves, ...



Introduction to Scientific Computing
Lesson 11: Parallelization



Slide 8

Performance Evaluation 1

- performance evaluation of algorithms and computers
 - quality and performance of parallel computer
 - quality and performance of parallel(ized) algorithm

- p processors of (single) performance l, overall work W:

$$W(p) = l \cdot \sum_{i=1}^p i \cdot t_i, \quad t_i : \text{time with } i \text{ processors busy}$$

- average parallelism:

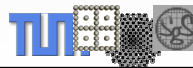
$$A(p) = \frac{\sum_{i=1}^p i \cdot t_i}{\sum_{i=1}^p t_i} = \frac{1}{l} \cdot \frac{W(p)}{\sum_{i=1}^p t_i}$$

- several pessimistic estimates:

$$A(p) = \log_2(p)$$

- speedup S:

$$S = \frac{T(1)}{T(p)} \quad \begin{array}{l} \text{time sequential algorithm on one processor} \\ \text{time parallel algorithm on } p \text{ processors} \end{array}$$



Performance Evaluation 2

- efficiency E: $E = \frac{S}{p} = \frac{T(1)}{p \cdot T(p)}$

- stronger definition: always use *best* sequential algorithm for comparison!

- usually $S < p$ and $E < 1$; but: competition!

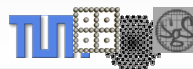
- Amdahl's Law:

- assumption: each program has some part $0 \leq \text{seq} < 1$ that can only be treated in a sequential way

$$S \leq \frac{1}{\text{seq} + \frac{1 - \text{seq}}{p}} \leq \frac{1}{\text{seq}}$$

- already $\text{seq} = 0.01$ limits S to 100

- cf. LINPACK: typically $\text{seq} < 0.1$



Performance Evaluation 3

➤ model of Gustafson:

- runtime on p processors normed to 1, there a sequential part seq ; hence

$$T(1) = seq + p \cdot (1 - seq), \quad S \leq T(1) = p + seq \cdot (1 - p)$$

- difference to Amdahl: sequential part w.r.t. to runtime on 1 processor decreases with increasing p (often realistic)

➤ another important quantity: CCR

- Communication-to-Computation Ratio
- Big CCRs cause problems!
- CCR often increases with increasing p and constant problem size (example: iterative methods for $Ax=b$)
- therefore: do not compare speedups for different p , but same problem size

